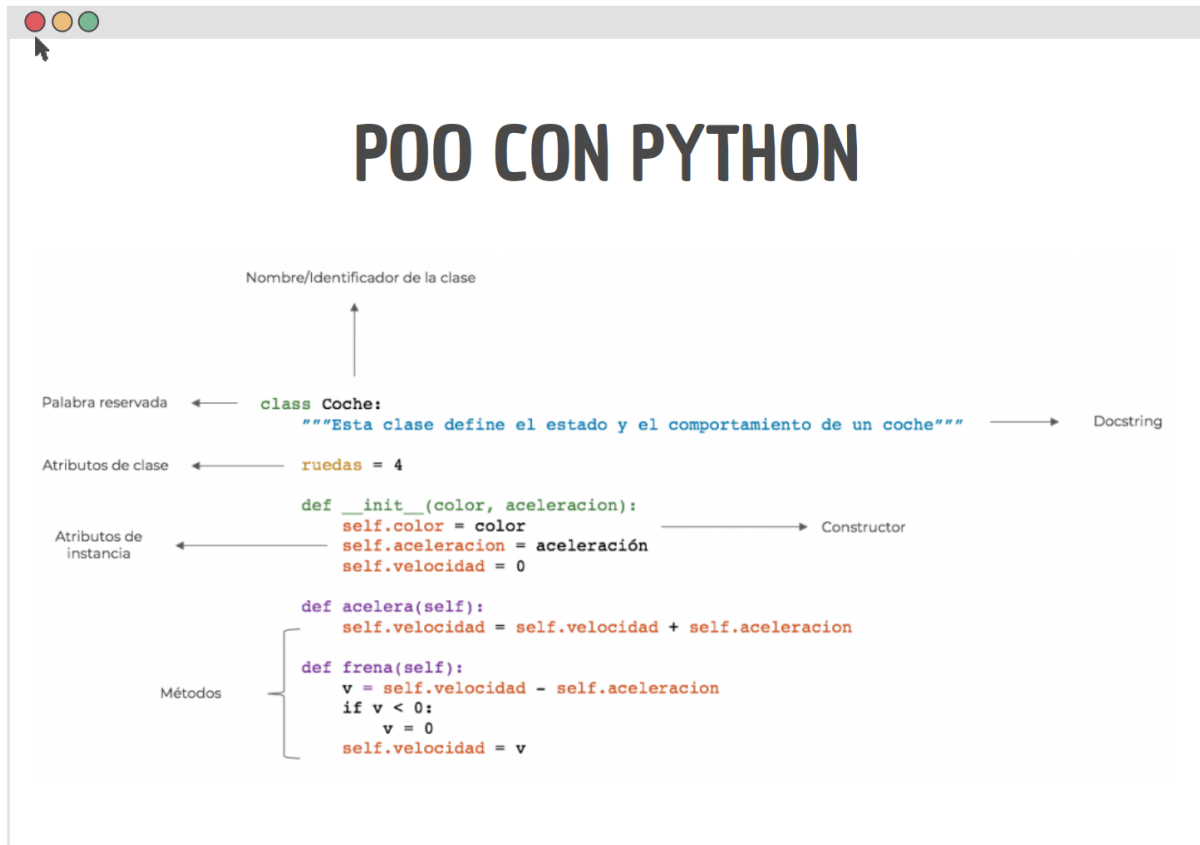




PROGRAMACIÓN WEB 2



Profesor(a):
Quispe Cruz, Marcela

Estudiantes:
Arias Quispe, Jhonatan David
Mamani Huarsaya, Jorge Luis
Velarde Saldaña Jhossep Fabritzio

Repositorio GitHub:
<https://github.com/jorghee/OOP-with-python>

28 de mayo, 2024

El programa Agenda telefónica

- El programa necesita guardar los datos que el usuario dispone, estos datos deben ser almacenados en algún tipo de archivos. Debido a que se necesitan guardar listas de objetos, se ha decidido hacer uso de archivos JSON, un formato ligero de intercambio de datos, fácil de leer y escribir con la biblioteca json de Python, adecuado para datos estructurado.
- Leer y escribir en archivos con Python viene de forma nativa y se utiliza a través de las funciones integradas `open`, `read`, `write` entre otras. Sin embargo para manejar ciertos formatos específicos como JSON, Python ofrece el módulo estándar json.

La clase Contacto

Contiene los campos mencionados necesarios para identificar al contacto

```
1 def __init__(self, nombre: str, telefono: str, direccion: str, relacion: str):
2     self._nombre = nombre
3     self._telefono = telefono
4     self._direccion = direccion
5     self._relacion = relacion
```

Además debe de contener los métodos que se llaman automáticamente al momento de representar como string al objeto actual.

```
1 def __repr__(self) -> str:
2     return self.__str__()
3
4 def __str__(self) -> str:
5     return (f"\nNombre:\t\t{self._nombre}\nTelefono:\t\t{self._telefono}\n"
6           f"Direccion:\t\t{self._direccion}\nRelacion:\t\t{self._relacion}\n")
```

Debemos de tener un método que nos permita parsear el objeto actual como un diccionario. Por ejemplo, que el identificador de referencia al campo name sea como clave y su valor sea el correspondiente valor del campo.

```
1 def parsear_a_json(self):
2     return {
3         "nombre": self.nombre,
4         "telefono": self.telefono,
5         "direccion": self.direccion,
6         "relacion": self.relacion
7     }
```

Un método que reciba como argumento un diccionario, básicamente es el objeto JSON parseado por el módulo json de Python. Este diccionario contiene los campos de una instancia de la clase Contacto, por lo tanto con estos datos se debe crear el nuevo objeto. Si analizamos, este método será parte de cargar los datos guardados en un archivo JSON.

```
1 @staticmethod
2 def parsear_de_json(dic):
3     return Contacto(dic["nombre"], dic["telefono"], dic["direccion"], dic["relacion"])
```

El método debe de ser estático porque necesitamos acceder a este mismo sin tener que crear aún un objeto Contacto.

La clase Agenda

Esta clase contiene un campo `self._contacts` que es una lista de objetos Contacto. Aquí tenemos una particularidad a destacar. Se decidió que el **archivo que guarda los contactos se recupera automáticamente** al momento de crear una instancia de esta clase.

```
1 def __init__(self, path_data):
2     self._contacts = self.recuperar_agenda(path_data)
```

Por lo tanto, comenzaremos explicando el método `recuperar_agenda()`.

El método `recuperar_agenda(path_data)`

La lógica es recuperar el archivo con la dirección que nosotros como desarrolladores han decidido, en caso no exista el archivo, entonces se regresa una lista vacía; de lo contrario necesitamos cargar el archivo.

```
1 def recuperar_agenda(self, path_data):
2     if not (os.path.exists(path_data)):
3         return []
4
5     with open(path_data, "r") as data:
6         datos = json.load(data)
7         print("Recuperación exitosa...\n")
8     return [Contacto.parsear_de_json(dic) for dic in datos]
```

Cuando cargamos el archivo, este se carga en diccionarios, ya que la conversión de objetos JSON es a diccionarios en Python. Entonces aquí debemos de utilizar el método `parsear_de_json()` que recibe el diccionario y crear con estos valores el objeto Contacto.

El método `getContact(pattern)`

Básicamente la lógica consiste en iterar sobre la lista `self._contacts` y recuperar su campo `nombre`.

Ahora necesitamos comparar dicho nombre con el patrón ingresado. Para resolver este problema se ha utilizado el siguiente algoritmo.

Algoritmo de Knuth-Morris-Pratt (KMP): El algoritmo KMP utiliza la información del propio patrón para evitar retrocesos innecesarios en la cadena de texto. Esto se logra mediante la construcción de un arreglo de prefijo (función de prefijo o función π), también conocido como "arreglo de fallos" o "tabla de fallos", que se utiliza para saltar comparaciones redundantes.

```
KMP-MATCHER( $T, P, n, m$ )
1   $\pi$  = COMPUTE-PREFIX-FUNCTION( $P, m$ )
2   $q = 0$  // number of characters matched
3  for  $i = 1$  to  $n$  // scan the text from left to right
4      while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
5           $q = \pi[q]$  // next character does not match
6      if  $P[q + 1] == T[i]$ 
7           $q = q + 1$  // next character matches
8      if  $q == m$  // is all of  $P$  matched?
9          print "Pattern occurs with shift"  $i - m$ 
10      $q = \pi[q]$  // look for the next match

COMPUTE-PREFIX-FUNCTION( $P, m$ )
1  let  $\pi[1 : m]$  be a new array
2   $\pi[1] = 0$ 
3   $k = 0$ 
4  for  $q = 2$  to  $m$ 
5      while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
6           $k = \pi[k]$ 
7      if  $P[k + 1] == P[q]$ 
8           $k = k + 1$ 
9       $\pi[q] = k$ 
10 return  $\pi$ 
```

Figure 1: Pseudocódigo del algoritmo KMP

Entonces en ves de devolver el índice de la primera posición de la cadena de texto de todas las coincidencias, nosotros hemos hecho que devuelva **True** o **False**.

```
1  # Algoritmo Knuth-Morris-Pratt (KMP)
2  def _match(self, name, pattern):
3      lps = self._compute_lps(pattern)
4      i = 0
5      j = 0
6
7      while i < len(name):
8          if pattern[j] == name[i]:
9              i += 1
10             j += 1
11
12             if j == len(pattern):
13                 j = lps[j - 1]
14                 return True
15             elif i < len(name) and pattern[j] != name[i]:
16                 if j != 0:
17                     j = lps[j - 1]
18                 else:
19                     i += 1
20             return False
21
```

```
22 # Generación de la función LPS
23 def _compute_lps(self, pattern):
24     lps = [0] * len(pattern)
25     length = 0
26     i = 1
27
28     while i < len(pattern):
29         if pattern[i] == pattern[length]:
30             length += 1
31             lps[i] = length
32             i += 1
33         else:
34             if length != 0:
35                 length = lps[length - 1]
36             else:
37                 lps[i] = 0
38                 i += 1
39     return lps
```

El método guardar_agenda()

El método consiste en iterar en los objetos de la lista `self._contacts` y generar una lista ya no de objetos `Contacto`, sino ahora una lista de diccionarios, usando el método `parsear_a_json()` de la clase `Contacto`.

```
1 def guardar_agenda(self):
2     contacts_pars = [contact.parsear_a_json() for contact in self._contacts]
3     with open("./data/data.json", "w", encoding = "utf-8") as save:
4         json.dump(contacts_pars, save, ensure_ascii=False, indent=2)
5         print("Guardacion exitosa...\n")
```

Método updateContact

Definición del Método

```
1 def updateContact(self, newContact):
2     for idx, saved in enumerate(self._contacts):
3         if saved.nombre == newContact.nombre:
4             self._contacts[idx] = newContact
5             print("El contacto se actualizó exitosamente...\n")
6             return
7     print("No se encontró el contacto a actualizar...\n")
```

Descripción del Funcionamiento

- **Propósito:** Actualizar un contacto existente en la lista `_contacts`.
- **Parámetros:** `newContact`: Un objeto de contacto que contiene la nueva información que se desea actualizar.
- **Flujo de Control:**
 - **Iteración:** El método recorre la lista `_contacts` usando `enumerate` para obtener tanto el índice como el objeto de contacto actual.
 - **Comparación:** Dentro del bucle, compara el nombre (`nombre`) del contacto guardado (`saved`) con el nombre del nuevo contacto (`newContact`).
 - **Actualización:** Si encuentra una coincidencia, reemplaza el contacto existente en la lista con el nuevo contacto (`newContact`).
 - **Salida:** Imprime un mensaje de éxito y sale del método con `return` para evitar iteraciones innecesarias.
 - **No Encontrado:** Si no se encuentra una coincidencia tras iterar toda la lista, imprime un mensaje indicando que no se encontró el contacto.

Método deleteContact

Definición del Método

```
1 def deleteContact(self, pattern):
2     for contact in self._contacts:
3         name = contact.nombre
4         isName = self._match(name.lower(), pattern.lower())
5         if isName:
6             self._contacts.remove(contact)
7             print("El contacto se eliminó exitosamente...\n")
8             return
9     print("No se encontró el contacto a eliminar...\n")
```

Descripción del Funcionamiento

- **Propósito:** Eliminar un contacto de la lista `_contacts` que coincida con un patrón dado.
- **Parámetros:** `pattern`: Un patrón de texto que se usará para buscar coincidencias en los nombres de los contactos.

- **Flujo de Control:**

- **Iteración:** El método recorre la lista `_contacts`.
- **Comparación:** Convierte los nombres de los contactos y el patrón a minúsculas para una comparación insensible a mayúsculas/minúsculas.
- **Coincidencia:** Utiliza un método auxiliar `_match` para verificar si el nombre coincide con el patrón.
- **Eliminación:** Si encuentra una coincidencia, elimina el contacto de la lista `_contacts`.
- **Salida:** Imprime un mensaje de éxito y sale del método con `return`.
- **No Encontrado:** Si no se encuentra una coincidencia tras iterar toda la lista, imprime un mensaje indicando que no se encontró el contacto.

El programa Carrito de Compras

La clase ProductStock

Por el momento solo contiene los 3 campos name, value y quantity, además de su correspondiente constructor y los métodos setters y getters.

```
1 class ProductStock:
2     def __init__(self, name, value, quantity):
3         self._name = name
4         self._value = value
5         self._quantity = quantity
```

La clase StockProducts

Contiene un campo que almacena los productos en stock. Por lo tanto puede ser una lista de objetos ProductStock, sin embargo, podemos especificar un id para cada tipo de producto, una forma de identificarlo unicamente y que con id directamente podamos acceder a la informacion del producto como es el nombre, costo y cantidad.

Qué estructura usar

Los diccionario en Python nos facilitaria acceder directamente a la informacion del producto sin la necesidad de tener que iterar por cada objeto ProductStock y comparar su campo name como se tendria que hacer si la estructura sería una simple lista de objetos ProductStock.

```
1 class StockProducts:
2     def __init__(self):
3         self._products = {}
```

El método add_product()

Se encargará de agregar un nuevo producto al diccionario, donde la clave es el nombre del producto, así ingresar a la información del producto es más eficiente.

```
1 def add_product(self, product):
2     self._products[product.name] = product
3     print("Producto agregado en stock exitosamente...")
```

El método get_product()

Se encargará de obtener el producto según el nombre que reciba como argumento, aquí podemos observar la funcionalidad del diccionario, ya que solo necesitamos pasar el argumento como clave para buscar en el diccionario en vez de tener que iterar por cada objeto en caso de una lista. Claro esta que son estructuras de datos y su concepto no involucra saber cómo esta implementado por dentro.

```
1 def get_product(self, name):
2     return self._products.get(name)
```

Como vemos, solo consta en usar la función `get()` en lugar de acceder nativamente al valor del diccionario. Si nosotros usamos esta forma, entonces si en futuras verificaciones como es en el caso de la clase **ShoppingCart** puede arrojar un error al no encontrar dicha clave.

El método `delete_products()`

Este método es usado en el método `finalize_purchase()` de la clase **ShoppingCart**. Se encarga de reducir la cantidad de unidades disponibles de cada producto que esta en stock.

```
1 def delete_products(self, name, quantity):
2     if self._products[name].quantity >= quantity:
3         self._products[name].quantity -= quantity
4         print("Producto eliminado exitosamente...")
5     else:
6         print("Producto no encontrado...")
```

La clase **ShoppingCart**

Al igual que la clase **StockProducts** usamos el concepto de **Asociación**, pues se tendrá una referencia una instancia de **StockProducts**, lo cual nos permite saber la información de la lista de productos en stock, claro está que es un diccionario.

Almacenar los productos y la cantidad de unidades en el carrito de compras

Para poder conectar con los productos en stock rápidamente podemos pensar que esta estructura también debe de ser un diccionario donde la clave es el nombre del producto y el valor es la cantidad de unidades puestas en el carrito de compras

```
1 class ShoppingCart:
2     def __init__(self, stock):
3         self._stock = stock    # Productos en stock
4         self._item = {}       # Productos en el carrito
```

El método `add_item()`

Como vemos, este método debe de recibir 2 argumentos, que son el nombre del producto y la cantidad de unidades a comprar. Entonces la lógica en este método se divide en 2 partes:

- Verificar si el producto está en el campo que referencia a una instancia de **StockProducts** y verificar también si la cantidad a comprar es menor o igual que la cantidad disponible.
- Verificar que si el producto ya está en el carrito de compras solo aumentar la cantidad de unidades, de lo contrario agregar el producto al carrito de compras con su correspondiente cantidades de unidades iniciales.

```
1 def add_item(self, name, quantity):
2     if self._stock.get_product(name) and quantity <=
3         self._stock.get_product(name).quantity:
4         if self._item.get(name):
5             self._item[name] += quantity
6         else:
7             self._item[name] = quantity
8     else:
9         print("Producto no disponible...\n")
```

El método `finalize_purchase()`

Este método debe confirmar la compra modificando la referencia a `StockProducts` justamente la cantidad disponible de cada producto. Recordemos que podemos no confirmar y entonces la cantidad disponible para los otros usuarios no cambiará.

```
1 def finalize_purchase(self):
2     for key in self._item:
3         self._stock.delete_products(key, self._item[key])
4     self._item = {}
5     print("Compra finalizada...\n")
```

El método `calculate_total()`

Usando la clave de los diccionarios, podemos acceder al costo del producto y en la referencia a `StockProducts` y acceder a la cantidad comprada en el campo que se encargará de almacenar los productos agregados al carrito de compras. Sumamos esta multiplicación y ese sería el resultado final.

```
1 def calculate_total(self):
2     total = 0
3     for key in self._item:
4         total += self._stock.get_product(key).value * self._item[key]
5     return total
```